📚 Guide de Maîtrise des Principes SOLID

1. Introduction à SOLID

SOLID est un ensemble de cinq principes de conception orientée objet qui permettent d'écrire un code maintenable, évolutif et facile à comprendre.

- **S** - Single Responsibility Principle (SRP)
- **O** - Open/Closed Principle (OCP)
- **L** - Liskov Substitution Principle (LSP)
- **I** - Interface Segregation Principle (ISP)
- **D** - Dependency Inversion Principle (DIP)

S - Single Responsibility Principle (SRP)

Le **principe de responsabilité unique** stipule qu'une classe ou un module doit avoir une seule responsabilité, c'est-à-dire qu'il doit être en charge d'une seule partie du comportement d'un programme. Cela signifie que si une classe ou un module a plusieurs raisons de changer, il enfreint ce principe. Ce principe aide à réduire le couplage et à améliorer la maintenabilité du code, car les modifications dans une fonctionnalité n'affecteront pas d'autres parties du programme.

O - Open/Closed Principle (OCP)

Le **principe ouvert/fermé** stipule que les entités du logiciel (classes, modules, fonctions, etc.) doivent être ouvertes à l'extension, mais fermées à la modification. Cela signifie que vous devez pouvoir ajouter de nouvelles fonctionnalités sans modifier le code existant. L'extension du comportement doit se faire par l'ajout de nouvelles classes ou modules, ce qui permet de maintenir la stabilité du système tout en permettant son évolution.

L - Liskov Substitution Principle (LSP)

Le **principe de substitution de Liskov** stipule que les objets d'une classe dérivée doivent pouvoir être utilisés comme des objets de la classe de base sans altérer le comportement correct du programme. Autrement dit, une sous-classe doit être substituable à sa classe parente sans que cela ne provoque de dysfonctionnement ou de comportements inattendus dans le système. Cela garantit que l'héritage ne viole pas les principes de fonctionnement du programme.

I - Interface Segregation Principle (ISP)

Le **principe de séparation des interfaces** stipule qu'il est préférable de créer des interfaces spécifiques et petites plutôt que de créer une interface générale. Autrement dit, les clients d'une interface ne doivent pas être contraints à dépendre de méthodes qu'ils n'utilisent pas. Ce principe aide à éviter que des classes implémentent des méthodes inutiles qui ne sont pas pertinentes pour leur propre fonctionnement, ce qui rend le code plus flexible et plus compréhensible.

D - Dependency Inversion Principle (DIP)

Le **principe d'inversion des dépendances** stipule que les modules de haut niveau (qui définissent des règles métiers ou des processus) ne doivent pas dépendre des modules de bas niveau (qui contiennent des détails d'implémentation), mais les deux doivent dépendre d'abstractions (interfaces, classes abstraites). De plus, les abstractions ne doivent pas dépendre des détails, mais les détails doivent dépendre des abstractions. Cela permet de réduire le couplage entre les composants et facilite les tests, la maintenance et l'évolution du code.

Ces principes sont essentiels pour construire des systèmes logiciels robustes, flexibles et faciles à maintenir. Ils permettent de réduire la complexité et d'améliorer la qualité du code au fur et à mesure de son évolution.

2. Explication des Principes SOLID avec Exemples

🟤 1. Single Responsibility Principle (SRP) - Principe de responsabilité unique

🔹 **Définition** : Une classe ne doit avoir qu'une seule raison de changer. Chaque classe doit être responsable d'une seule chose.

❌ ***Mauvais exemple (Ne respecte pas SRP)***

```
public class Report {
    public void generateReport() {
        // Générer le rapport
    }

    public void printReport() {
```

```
        // Imprimer le rapport
    }

    public void saveToDatabase() {
        // Sauvegarder dans la base de données
    }
}
```

🔴 **Problème** : Cette classe a plusieurs responsabilités (générer, imprimer et sauvegarder).

☑️ *Bon exemple (Respecte SRP)*

```
public class ReportGenerator {
    public void generateReport() {
        // Générer le rapport
    }
}

public class ReportPrinter {
    public void printReport() {
        // Imprimer le rapport
    }
}

public class ReportSaver {
    public void saveToDatabase() {
        // Sauvegarder dans la base de données
    }
}
```

✓ **Solution** : On divise en plusieurs classes ayant chacune une seule responsabilité.

🟤 2. Open/Closed Principle (OCP) - Principe ouvert/fermé

🔹 **Définition** : Une classe doit être **ouverte à l'extension mais fermée à la modification**.

❌ *Mauvais exemple (Ne respecte pas OCP)*

```
public class PaymentProcessor {
    public void processPayment(String type) {
        if (type.equals("CreditCard")) {
            // Traiter le paiement par carte
        } else if (type.equals("PayPal")) {
            // Traiter le paiement via PayPal
        }
    }
}
```

🔴 **Problème** : Chaque fois qu'on ajoute un nouveau mode de paiement, on doit modifier la classe.

☑ Bon exemple (Respecte OCP)

```
public interface PaymentMethod {
    void pay();
}

public class CreditCardPayment implements PaymentMethod {
    public void pay() {
        // Traiter le paiement par carte
    }
}

public class PayPalPayment implements PaymentMethod {
    public void pay() {
        // Traiter le paiement via PayPal
    }
}

public class PaymentProcessor {
    public void processPayment(PaymentMethod method) {
        method.pay();
    }
}
```

✓ **Solution** : On utilise une interface et on ajoute de nouvelles implémentations sans modifier le `PaymentProcessor`.

🟤 3. Liskov Substitution Principle (LSP) - Principe de substitution de Liskov

🔹 **Définition** : Une sous-classe doit pouvoir être substituée à sa classe parente sans modifier le comportement du programme.

❌ *Mauvais exemple (Ne respecte pas LSP)*

```
public class Rectangle {
    protected int width, height;

    public void setWidth(int width) { this.width = width; }
    public void setHeight(int height) { this.height = height; }
}

public class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        this.width = this.height = width;
    }

    @Override
    public void setHeight(int height) {
        this.width = this.height = height;
    }
}
```

🔴 **Problème** : Square ne respecte pas le comportement attendu de `Rectangle` (modifier `width` ne doit pas modifier `height`).

☑️ *Bon exemple (Respecte LSP)*

```
public interface Shape {
    int getArea();
}

public class Rectangle implements Shape {
    protected int width, height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
```

```
    }

    public int getArea() {
        return width * height;
    }
}

public class Square implements Shape {
    private int side;

    public Square(int side) {
        this.side = side;
    }

    public int getArea() {
        return side * side;
    }
}
```

✓ **Solution** : On utilise une interface Shape pour éviter les problèmes d'héritage.

🟢 4. Interface Segregation Principle (ISP) - Principe de ségrégation des interfaces

🔹 **Définition** : Une interface ne doit contenir que les méthodes nécessaires à son utilisation.

❌ *Mauvais exemple (Ne respecte pas ISP)*

```
public interface Worker {
    void work();
    void eat();
}

public class Robot implements Worker {
    public void work() {
        // Travailler
    }

    public void eat() {
        throw new UnsupportedOperationException("Les robots ne
```

```
mangent pas!");
    }
}
```

🔴 **Problème** : Robot implémente une méthode qui ne lui correspond pas.

☑️ ***Bon exemple (Respecte ISP)***

```
public interface Workable {
    void work();
}

public interface Eatable {
    void eat();
}

public class Human implements Workable, Eatable {
    public void work() {
        // Travailler
    }

    public void eat() {
        // Manger
    }
}

public class Robot implements Workable {
    public void work() {
        // Travailler
    }
}
```

✓ **Solution** : On divise `Worker` en deux interfaces (`Workable` et `Eatable`).

🟤 5. Dependency Inversion Principle (DIP) - Principe d'inversion des dépendances

🔹 **Définition** : Les classes haut niveau ne doivent pas dépendre des classes bas niveau, mais d'abstractions.

❌ *Mauvais exemple (Ne respecte pas DIP)*

```
public class MySQLDatabase {
    public void connect() {
        // Connexion à MySQL
    }
}

public class Application {
    private MySQLDatabase database = new MySQLDatabase();

    public void start() {
        database.connect();
    }
}
```

🔴 **Problème** : `Application` dépend directement de `MySQLDatabase`, ce qui empêche d'utiliser un autre type de base de données.

☑️ *Bon exemple (Respecte DIP)*

```
public interface Database {
    void connect();
}

public class MySQLDatabase implements Database {
    public void connect() {
        // Connexion à MySQL
    }
}

public class PostgreSQLDatabase implements Database {
    public void connect() {
        // Connexion à PostgreSQL
    }
}

public class Application {
    private Database database;
```

```
    public Application(Database database) {
        this.database = database;
    }

    public void start() {
        database.connect();
    }
}
```

✓ **Solution** : On injecte une dépendance `Database` dans `Application`, permettant de changer facilement de base de données.